Python code

```

import numpy as np
import matplotlib.pyplot as plt


# ## **Question 1**


def func1(X):
    p1, p2 = 0.5, 0.5
    m1 = np.array([[1, -1]])
    m2 = np.array([[2, 3]])
    sigma1 = np.array([[2, 3], [3, 6.5]])
    sigma2 = np.array([[2, 3], [3, 6.5]])
    res = []
    for x in X:
        px1 = p1 / (np.linalg.det(sigma1)**0.5) * np.exp(-0.5 * (x - m1) @ np.linalg.inv(sigma1) @ (x - m1).T)
        px2 = p2 / (np.linalg.det(sigma2)**0.5) * np.exp(-0.5 * (x - m2) @ np.linalg.inv(sigma2) @ (x - m2).T)
        res.append(px1 / px2)
    return np.array(res)

def func2(X):
    p1, p2 = 0.1, 0.9
    m1 = np.array([[1, -1]])
    m2 = np.array([[2, 3]])
    sigma1 = np.array([[2, 3], [3, 6.5]])
    sigma2 = np.array([[2, 3], [3, 6.5]])
    res = []
    for x in X:
        px1 = p1 / (np.linalg.det(sigma1)**0.5) * np.exp(-0.5 * (x - m1) @ np.linalg.inv(sigma1) @ (x - m1).T)
        px2 = p2 / (np.linalg.det(sigma2)**0.5) * np.exp(-0.5 * (x - m2) @ np.linalg.inv(sigma2) @ (x - m2).T)
        res.append(px1 / px2)
    return np.array(res)


def plot_decision_boundary(training, label_train, func):
    # Total number of classes
    classes = np.unique(label_train)
    nclass = len(classes)

    class_names = []
    for c in classes:
        class_names.append('Class ' + str(int(c)))

    # Set the feature range for plotting
    max_x1 = np.ceil(np.max(training[:, 0])) + 1.0
    min_x1 = np.floor(np.min(training[:, 0])) - 1.0
    max_x2 = np.ceil(np.max(training[:, 1])) + 1.0
```

```
    min_x2 = np.floor(np.min(training[:, 1])) - 1.0

    xrange = (min_x1, max_x1)
    yrange = (min_x2, max_x2)

    # step size for how finely you want to visualize the decision boundary.
    inc = 0.05

    # generate grid coordinates. This will be the basis of the decision boundary visualization.
    (x1, x2) = np.meshgrid(np.arange(xrange[0], xrange[1] + inc / 100, inc),
                           np.arange(yrange[0], yrange[1] + inc / 100, inc))

    # size of the (x1, x2) image, which will also be the size of the
    # decision boundary image that is used as the plot background.
    image_size = x1.shape
    # make (x1, x2) pairs as a bunch of row vectors.
    grid_2d = np.hstack((x1.reshape(x1.shape[0] * x1.shape[1], 1, order='F'),
                         x2.reshape(x2.shape[0] * x2.shape[1], 1, order='F')))

    pred_label = np.where(func(grid_2d) > 1, 1, 2)
    # reshape the idx (which contains the class label) into an image.
    decision_map = pred_label.reshape(image_size, order='F')

    # create fig
    fig, ax = plt.subplots()
    ax.imshow(decision_map, vmin=np.min(classes), vmax=9, cmap='Pastel1',
              extent=[xrange[0], xrange[1], yrange[0], yrange[1]],
              origin='lower')

    # plot the class training data.
    data_point_styles = ['rx', 'bo', 'g*']
    for i in range(nclass):
        ax.plot(training[label_train == classes[i], 0],
                training[label_train == classes[i], 1],
                data_point_styles[int(classes[i]) - 1],
                label=class_names[i])
    ax.legend()

    plt.tight_layout()
    plt.show()

    return fig


training = np.array([[4, -1], [1, 3]])
label_train = [1, 2]
fig = plot_decision_boundary(training, label_train, func1)
fig.savefig("./figs/1d1.png", dpi=200)


training = np.array([[4, -1], [1, 3]])
label_train = [1, 2]
fig = plot_decision_boundary(training, label_train, func2)
```

```
fig.savefig("./figs/1d2.png", dpi=200)


# ## **Question 2**


def phi(x, h):
    X1 = np.array([0, 0.4, 0.9, 1.0, 6.0, 8.0])
    X2 = np.array([2.0, 4.0, 4.5, 5.0, 5.8, 6.7, 7.0])
    x1 = x - X1
    x2 = x - X2
    return np.where((-h <= x1) & (x1 <= h), 1, 0), np.where((-h <= x2) & (x2 <= h), 1, 0)

def density(x, h):
    res1, res2 = phi(x, h)
    n1 = res1.shape[0]
    n2 = res2.shape[0]
    return 1/n1 * np.sum(res1), 1/n2 * np.sum(res2)

x = np.linspace(-2, 10, 1201)
h = 0.5
P1 = []
P2 = []
for i in range(len(x)):
    p1, p2 = density(x[i], h)
    P1.append(p1)
    P2.append(p2)

_ = plt.plot(x, P1, label='P(x|S_1)')
_ = plt.plot(x, P2, label='P(x|S_2)')
_ = plt.legend()
_ = plt.xlabel("X")
_ = plt.ylabel("Prob")
_ = plt.title("Kernel Density Estimation (h={})".format(h))
plt.savefig("./figs/kde_{}.png".format(h))


x = np.linspace(-2, 10, 1201)
h = 1
P1 = []
P2 = []
for i in range(len(x)):
    p1, p2 = density(x[i], h)
    P1.append(p1)
    P2.append(p2)

_ = plt.plot(x, P1, label='P(x|S_1)')
_ = plt.plot(x, P2, label='P(x|S_2)')
_ = plt.legend()
_ = plt.xlabel("X")
_ = plt.ylabel("Prob")
_ = plt.title("Kernel Density Estimation (h={})".format(h))
plt.savefig("./figs/kde_{}.png".format(h))


x = np.linspace(-5, 15, 2001)
h = 2
```

```python
P1 = []
P2 = []
for i in range(len(x)):
    p1, p2 = density(x[i], h)
    P1.append(p1)
    P2.append(p2)

_ = plt.plot(x, P1, label='P(x|S_1)')
_ = plt.plot(x, P2, label='P(x|S_2)')
_ = plt.legend()
_ = plt.xlabel("X")
_ = plt.ylabel("Prob")
_ = plt.title("Kernel Density Estimation (h={})".format(h))
plt.savefig("./figs/kde_{}.png".format(h))


def decision_rule(x, h):
    res1, res2 = phi(x, h)
    return int(np.sum(res1) / np.sum(res2) > 1)

x = np.linspace(-2, 10, 1201)
h = 0.5
y = []
for i in range(len(x)):
    y.append(decision_rule(x[i], h))

_ = plt.scatter(x, y, s=0.1)
_ = plt.yticks([0, 1])
_ = plt.xlabel("X")
_ = plt.ylabel("class")
_ = plt.title("Decision Boundary")
plt.savefig("./figs/dc_h_{}.png".format(h))
plt.show()

x = np.linspace(-2, 10, 1201)
h = 1
y = []
for i in range(len(x)):
    y.append(decision_rule(x[i], h))

_ = plt.scatter(x, y, s=0.1)
_ = plt.yticks([0, 1])
_ = plt.xlabel("X")
_ = plt.ylabel("class")
_ = plt.title("Decision Boundary")
plt.savefig("./figs/dc_h_{}.png".format(h))
plt.show()

x = np.linspace(-5, 15, 2001)
h = 2
y = []
for i in range(len(x)):
    y.append(decision_rule(x[i], h))

_ = plt.scatter(x, y, s=0.1)
_ = plt.yticks([0, 1])
_ = plt.xlabel("X")
_ = plt.ylabel("class")
```

```
_ = plt.title("Decision Boundary")
plt.savefig("./figs/dc_h_{}.png".format(h))
plt.show()


# ## **Question 3**


X = np.array([-0.9, -0.7, -0.5, -0.3, -0.1, 0.1, 0.3, 0.5, 0.7, 0.9])
y = X**2
X, y


x1 = 0.0
d = np.abs(X - x1)
idx = np.argsort(d)[:4]
dmax = np.sort(d)[5]
weight = 1 - d / dmax
y1 = np.sum(weight[idx] * y[idx]) / np.sum(weight[idx])
weight
y1


x2 = 0.4
d = np.abs(X - x2)
idx = np.argsort(d)[:4]
dmax = np.sort(d)[5]
weight = 1 - d / dmax
y2 = np.sum(weight[idx] * y[idx]) / np.sum(weight[idx])
weight
y2


0.5*((y1-0)**2+(y2-0.16)**2)
```